Maintenance Strategies for Design Recovery and Reengineering

Volume 3
Methods

By: Dennis Braley and Allan Plumb

June 1990
Review Copy

Software Technology Branch
Johnson Space Center
National Aeronautics and Space Administration
Houston, TX 77058

ACKNOWLEDGEMENTS

The authors wish to thank participants from The MITRE Corporation who helped in the preparation of this document: Lois Morgan, who merged and edited several unpublished documents to form this document, Debra McGrath, who reviewed all volumes and continues to contribute to the design of the environment, and Dona Erb, who also reviewed all volumes.

CONTENTS

CONTENTS (concluded)

FIGURES

TABLES

1.0 INTRODUCTION

Programs in use today generally have all of the functional and information processing capabilities required to do their specified job. However, older programs usually use obsolete technology, are not integrated properly with other programs, and are difficult to maintain. Reengineering is becoming a prominent discipline as organizations try to move their systems to more modern and maintainable technologies. Johnson Space Center's (JSC) Software Technology Branch (STB) is researching and developing a system to support reengineering older FORTRAN programs into more maintainable forms that can also be more readily translated to a modern language such as FORTRAN 8x, Ada, or C. This activity has led to the development of maintenance strategies for design recovery and reengineering. These strategies include a set of standards, a methodology, and the concepts for a software environment to support design recovery and reengineering.

These products and concepts are documented in a five volume report, *Maintenance Strategies for Design Recovery and Reengineering*, of which this is the third volume. Volume 1, *Executive Summary and Problem Statement*, contains a statement of the problem and an overview of the STB's approach to solving the problem. Volume 2, *FORTRAN Standards*, defines new FORTRAN standards to augment the standards already in place in the mission planning and analysis domain at JSC. Volume 3, *Methods*, describes the methodology, which is based on experience in reengineering. This volume contains the "how-to" of the maintenance strategies presented in the first two volumes. Although the methods presented here can be performed manually, the manual process can be tedious and error prone. Existing tools in the STB's tool set that support some of the steps of a method are identified. At the present time these tools are available as stand-alone tools. Volume 4, *Concepts for an Environment*, presents the concepts and proposed architecture for an integrated environment to support the standards and methods. The proposed environment will integrate the existing tools, additional tools to be developed in-house, and commercial-off-the-shelf (COTS) tools. It will provide an improved, consistent user interface and will integrate the data that is generated and shared by the tools. Volume 5, *A Method for Conversion of FORTRAN Programs*, records the lessons learned in a pilot project that converted a large FORTRAN program to Ada. The reader who is interested in converting a program from FORTRAN to another language is encouraged to read Volume 5, which shares the lessons learned in performing a conversion, in order to fully understand the rationale behind the methods for conversion.

1.1 Approach

Reengineering is the combination of "reverse engineering" a working software system and then "forward engineering" a new system based on the results of the reverse engineering. Forward engineering is the standard process of generating software from "scratch." It is composed of the life cycle phases such as requirements, architectural design, detailed design, code development, testing, etc. In each phase, certain products are required and the activities which produce them are defined. Each product is required to be complete and consistent. To progress forward to a new phase normally requires a new representation of the products that involve more detail such as new derived requirements, design decisions, trade off evaluation between alternative approaches, etc. Finally, code is developed, which is the most complete, consistent, and detailed representation of the required product.

Reverse engineering is the reverse of forward engineering. It is the process of starting with existing code and going backward through the software development life cycle. Reverse engineering starts with the most detailed representation, which has also proven to be complete and consistent since it can currently do the job required. Developing products in reverse involves abstracting out only the essential information and hiding the non-essential details at each reverse step. Life cycle products are, therefore, obtained by abstracting from more detailed representations to more abstract ones. This process should proceed faster than forward engineering since all of the details required are available.

How far to go backward in the reverse engineering process before it is stopped and forward engineering begins is a critical question and involves trade offs. It is important to understand all of what the program does, all of the information it handles, and the control flow since these are probably required to get the job done. This implies taking the reverse process far enough to understand what the "as is" program is. Figure 1 illustrates reengineering with full reverse engineering to the point of the recovery of requirements and to the point of recovery of the design of the program. Reverse engineering is referred to as "design recovery" when the reverse engineering process stops at the recovery of the design of the implementation, rather than proceeding on to a higher level of abstraction to include the recovery of the requirements.

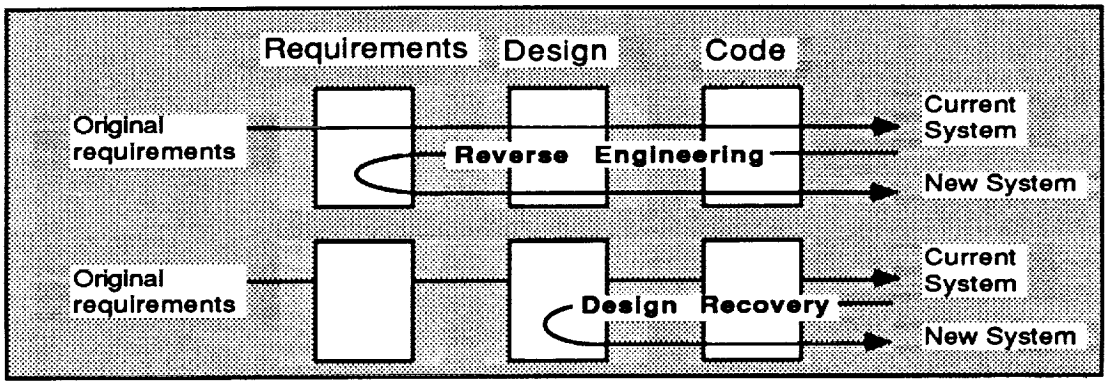

Figure 1. Levels of Reengineering

Design recovery involves recovering information about the code modules, the data structures, and their interrelationships in an existing program. This information supports the programmer/analyst who is maintaining an unfamiliar large FORTRAN program, upgrading it for maintainability, or converting it to another target language. However, a better job of redesigning a program can be accomplished with requirements recovery than with design recovery. To carry the reverse engineering process beyond design recovery to requirements recovery is difficult and requires higher levels of domain knowledge to do the abstractions. The "whys" of the requirements, design, and implementation can only be provided by someone very familiar with the program and the domain. This level of expertise is often very difficult to find and have dedicated to the reengineering process.

The philosophy of the reengineering process is to capture the total software implementation in an electronic form. This includes source code, data structures, documentation, etc. A progression in electronic forms can ensure that the total consistent and complete requirements are represented. By the continuing process of abstracting the information about the program

into different representations, the engineer can remain confident that information is not being lost. Table 1 shows the information that can be extracted from the source code of the current system for each of the three phases of the software development life cycle. Tools are already available in the STB tool set to obtain the information identified for the detailed design phase. Modifications to existing tools will yield the information identified with the preliminary design phases. However, the tools to support the abstraction to the level of requirements analysis must be developed or purchased as COTS products in the future.

Table 1. Information Extracted from the Current System by Reverse Engineering

REQUIREMENTS ANALYSIS	PRELIMINARY DESIGN	DETAILED DESIGN
• Data flow diagrams	• Definition of virtual packages • Abstraction of module dictionary	• Control structure (calling tree and control flow chart) • Module dictionary
• Entity-relationship diagrams	• Abstraction of data dictionary	• Data structure (cross-reference information) • Data dictionary
• Requirements	• Reproduced design document frame	• Documentation of "as-is" program (what it does and how it does it)

For this reason, initially the methods and tools developed by the STB assume reverse engineering only to the design recovery stage. That is as far as the STB's existing tools support. Additional software tools are needed to support the generation of the more abstract products required for reverse engineering as well as the capture of rationale and decisions of the engineer. Therefore, the STB is tracking emerging COTS tools and research that is being performed to push reverse engineering further back into the life cycle. The methods presented here, and the environment to support them, are designed to evolve as the supporting technologies mature. The current standards, methods, tools, and environment are all designed to be sufficiently flexible and extendible to enable the strategies to be extended to cover the full spectrum of reverse engineering. The evolution of these strategies depends upon the active participation and input from the JSC software maintenance community.

1.2 Contents of Volume

Figure 2 shows the upgrade paths supported by five of the methods that are defined in this volume. The boxes in the diagram denote four states, from right to left :

- Other target language	Program converted to FORTRAN 8x, C, or Ada.
- New "standard" FORTRAN	FORTRAN program that meets the standards proposed in Volume 2 to make a program easier to understand, to maintain, and to translate to another target language.
- COMGEN-compatible FORTRAN	FORTRAN program that meets the existing COMGEN standards, developed and followed in JSC's mission planning and analysis domain.
- Arbitrary FORTRAN	Any other FORTRAN program.

The maintainability of a program improves incrementally with upgrades from a program state at the left of the diagram to a state to its right. Budget, available manpower, type of maintenance problems being encountered, and other individual constraints will determine the maintainability upgrade path selected by the management of a given project. The basic approach and the tool set (although, not necessarily the individual tools) are the same for each of the upgrade methods; the difference is the current state of the subject FORTRAN program and the desired upgraded state.

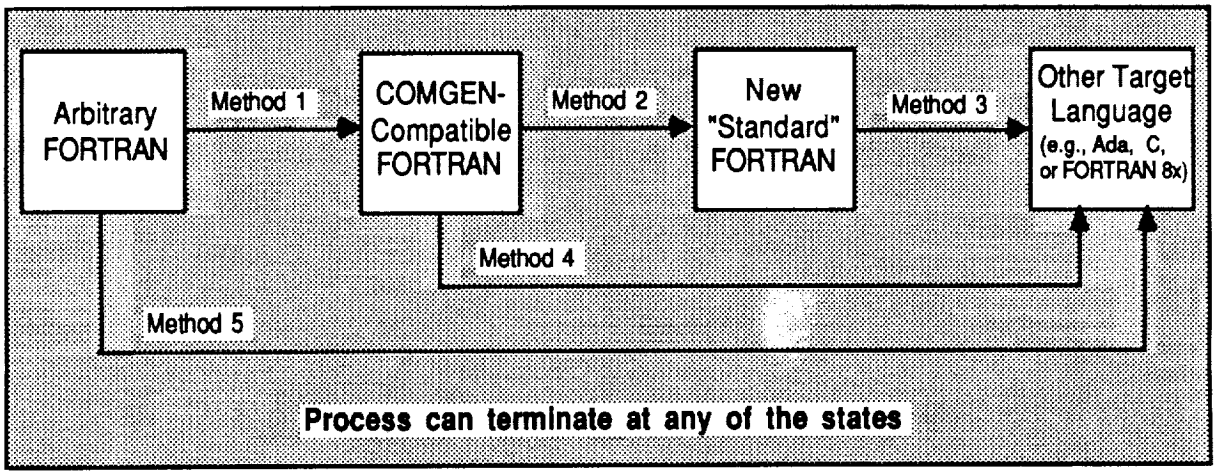

Figure 2. Alternative Paths for Improving Maintainability

In addition to the five methods identified in figure 2, a method for design recovery is defined in this volume. Design recovery is different from the activities addressed by the other methods, because the focus in design recovery is to gain an understanding of the subject program, not to change it in any way except to insert comment statements into the program to capture any previously undocumented knowledge. Some degree of design recovery is fundamental to understanding any large program that is being maintained or upgraded. Design recovery is the first step required for both Methods 4 and 5, which convert a program from a current state of FORTRAN to FORTRAN 8x, Ada, or C using structural and other major modifications to apply software engineering principles.

This volume is organized with the next section addressing a method for design recovery, followed by one section for each of the the five methods identified in figure 2 for improving the maintainability of a subject program. Following a brief conclusion section, an appendix provides summary information on STB's standard in-line documentation, tools, and data sets. A glossary of acronyms and terms and a list of references for the document are also provided.

2.0 METHOD FOR DESIGN RECOVERY

The method presented in this section represents one possible approach to design recovery. The approach is taken from the perspective of someone who is examining a large body of unfamiliar code and trying to understand it as a whole. A programmer/analyst who is concerned only in maintaining or enhancing a particular function can probably take a more localized approach if that person already understands the overall architecture of the program under analysis.

When first trying to understand a large program, the programmer/analyst can feel overwhelmed by the size and complexity of the program. Avoiding information inundation depends on techniques for maintaining control over the information being browsed. The primary goal of this proposed method for design recovery is to present ways to extract and manage the large body of information about the implemented design that is embedded in the code and its structure. The programmer/analyst must gain an understanding of the subject program as a whole, and as the sum of its parts. Because this process is somewhat intuitive, like putting together a jigsaw puzzle, it can not be totally automated. However, much of the knowledge that is contained in the source code can be extracted and organized by the use of the STB's software tools and strategies for design recovery. All of the tools identified in this volume are already in the STB's tool set. (See the appendix for a short description of each of the tools referenced in this volume.)

It is recommended that, as information is extracted in the process of design recovery, this knowledge should be captured and preserved in-line with the code by following the documentation standards established in the mission planning and analysis domain at JSC. This means using structured comment statements, referred to as "CD" statements, in the prolog of each subprogram as the repository to store the information. (See the appendix for brief descriptions of each of the CD statements. For further information, see the *Software Development and Maintenance Aids Catalog* (NASA IN 86-FM-27.)) Information about requirements (*what* the subprogram does) should be recorded in CD0 (identification) and CD1 (purpose) statements. Information about the internal design of a subprogram (*how* the subprogram does its job) should be recorded in CD2 through CD9 plus in header comment statements before blocks of code.

An ideal design recovery would recover the following information:

- Purpose, size, and scope of the program as a whole (i.e., overview).
- Subprogram interrelations (i.e., control structure).
- Interrelations among global data items and subprograms (i.e., data structure).

- Subprogram groupings (i.e., abstraction of the program structure to a higher level).
- Function of each subprogram (i.e., what each subprogram does).

2.1 Obtain an Overview

The first step in approaching a large, unfamiliar program is to try to get a feel for its purpose, size, and scope. The following steps are recommended:

- Find the purpose of the program. Read any available requirements and design documents, out of date or not. Read the prologs, if any, of the top level program units where the program purpose may be given, although usually only at a low level. The AUTODOC and SUBDOC processors provide documentation nicely formatted for browsing if the program is contains CD statements. Talk to someone who knows the program, if available.
- Estimate the size and scope of the program. Identify the main program and the number of subprograms; the number and size of source files; and the number, size, and type of COMMON blocks. Use the TOCGEN processor to generate an alphabetized list of subprograms with the lines-of-code count for each subprogram and the total lines of code for the program.

2.2 Identify Control Structure

The next step in understand an existing FORTRAN program usually is to learn about its control structure. The following steps are recommended:

- Generate a list of subprogram vs. subprograms called, using the CREATE processor.
- Generate a list of subprogram vs. subprograms called-by, using the CREATE processor.
- Generate a list of files vs. entry points, using the CREATE processor.
- Generate a list of entry points vs. files, using the CREATE processor.
- Generate hierarchical call graphs, using the DEPCHT and SETGEN processor.
    - -- Eliminate utility subprograms. Locate them by identifying the small subprograms that are called by several other subprograms, but do not call any other subprograms. Obtain a second call graph without the utilities to obtain an overview graph for the program.
    - -- Annotate the hierarchical call graph to show conditionals, loops, cases, etc. Look at the conditions that will cause each subprogram to be called. Determine if a subprogram is main line (i.e., always called), or called only if a certain flag is set. Determine what conditions will cause the flag to be set. This activity will help to identify the most important or, at least, most frequently used subprograms.

-- Start to notice the groupings. For example, subprogram A is called only by subprogram B, but subprogram C is called by several subprograms in different areas of the hierarchy.

2.3 Identify Data Structure

The data components of a program need to be identified and classified just as much as the control structures. In analyzing the program data to understand the data structure and the use of data items, the following procedure is recommended:

- Generate a list of COMMON blocks versus subprograms, using the CCREF processor.
- Generate list of all COMMON variables in a subprogram showing if the variable is initialized, set, or used in the subprogram, using the CREATE processor.
- Generate a list of COMMON variables versus subprograms, using the INVERT processor.
- Examine the COMMON blocks to gain understanding of their contents.
  - -- Determine if a COMMON block contains only single data items, groups of related data items, or groups of unrelated data items.
  - -- Determine the scope of each COMMON block, i.e., which subprograms reference the data items in a given COMMON block using the CCREF and RELREF processors. The scope of a COMMON block can help in analyzing the potential impact of making a maintenance modification or in deciding on packaging when improving the maintainability of a program.
  - -- Examine data item usage by subprogram to find the most often used parameters.
- Attempt to determine the meanings of the most used data items. To do this, examine these items in source code and read their descriptions, if available in the comments within the code and within the COMON database (CDB) for variables in COMMON blocks.
- Completely define the data structures containing the most frequently used data.
- Locate all of the input/output statements in the subject program.

For COMGEN-compatible FORTRAN programs, the COMMON database (CDB) is the best place to find and to store additional information about the structure of the global data. One of the primary advantages of COMGEN compatibility is that the structure of the COMMON blocks is defined in the CDB and maintained with the aid of various of the STB processors. The CDB is briefly described in the appendix and glossary, but for detailed information, see the *Software Development and Maintenance Aids Catalog* (NASA IN 86-FM-27). Much of the creation of a CDB can be automatically generated from the source code using the STB's tools as described in section 3, where the method for upgrading an arbitrary FORTRAN program to COMGEN-compatibility is described. Any data definitions that are missing from the CDB

should be added as they are obtained. In this way, knowledge that is obtained in the design recovery process is preserved for other persons who are assigned to maintain the software.

2.4 Abstract Program Structure to a Higher Level

The modules that make up an old FORTRAN program are seldom self-evident, since they are often coded not as a single source file, but rather as a group of separate files or subprograms. To abstract the structure of the subject program to a higher level of abstraction than the subprogram level, the job is to properly group these subprograms to form the *modules* that constitute the logical structure of the program (See figure 3). Program modules have three basic sets of characteristics: control cohesion, data cohesion, or organizational convention.

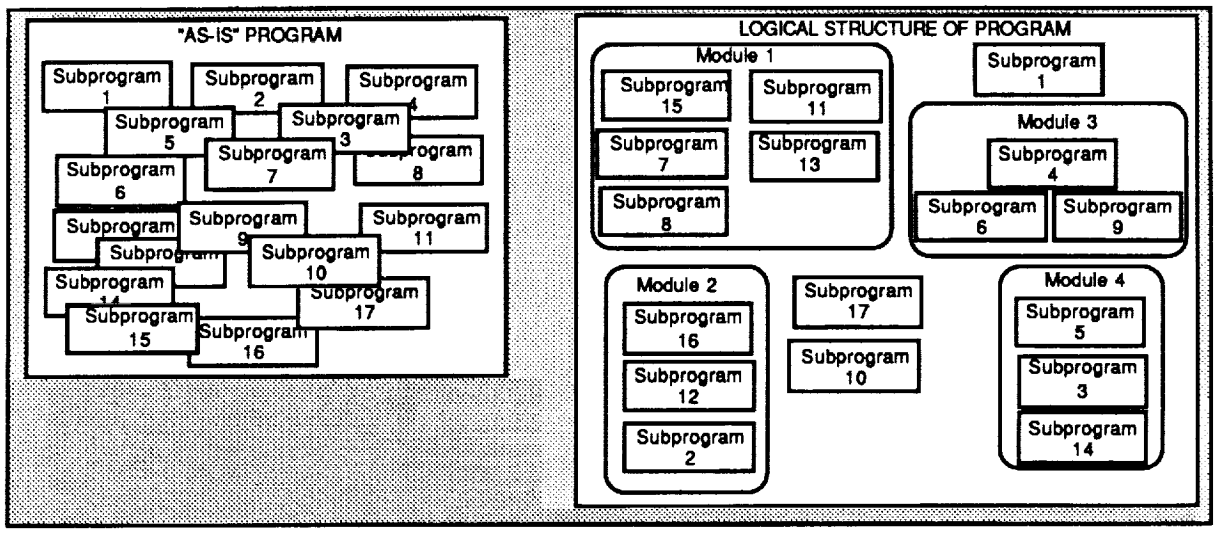

Figure 3. Grouping Subprograms into Modules

A module that exhibits control cohesion is a group of subprograms that is characterized by a single entry to the set of subprograms and by the property that all of the other subprograms in the module are reachable only from that single subprogram entry. A module of the data cohesion type is related because it operates on the same data, and thus is object-like, in the object-oriented sense. It is centered on a specific data entity (e.g., a screen window) and a set of subprograms that manage that entity (e.g., create a window or move a window). A module of the organizational convention type is organized as a group of subprograms that serves a single purpose, such as general purpose utilities or mathematical routines.

There will also be subprograms that do not fall into any of the above areas. These are primarily independent routines rather than modules. These can be classified into three groups: utility routines, domain-specific routines, top level program executive modules, and unclassified, independent, single purpose routines. Domain-specific routines are routines that are called from several places within a program. They are used in a manner similar to utility subprograms, but their purpose is not general enough for them to be considered as true utility routines. Data declaration files, such as BLOCK DATA routines in FORTRAN, are examples of the unclassified routines.

The previous two steps, which identified the control structure and the global data structure of the subject program, serve as preparation for this step of grouping logically related subprograms. The following activities are recommended:

- Locate control cohesion modules by examination of the branches in the call graph created by the DEPCHT processor. The programmer/analyst might wish to repeatedly drop identified modules and rebuild the call graph so that the unclassified areas can be viewed more clearly. For a large program the programmer/analyst may wish to generate call graphs for subsets of the code using the SETGEN processor.

- Identify data cohesion modules and organizational convention modules (if any) in the code by examining the code and data structures in the source files with multiple entry points. In addition, use the CCREF processor to identify subprograms that access the same COMMON blocks; these subprograms are candidates for data cohesion modules. If the program is COMGEN-compatible, this analysis may result in logically grouping data by breaking down single master arrays into multiple master arrays and nesting them in the CDB, as shown in figure 4.

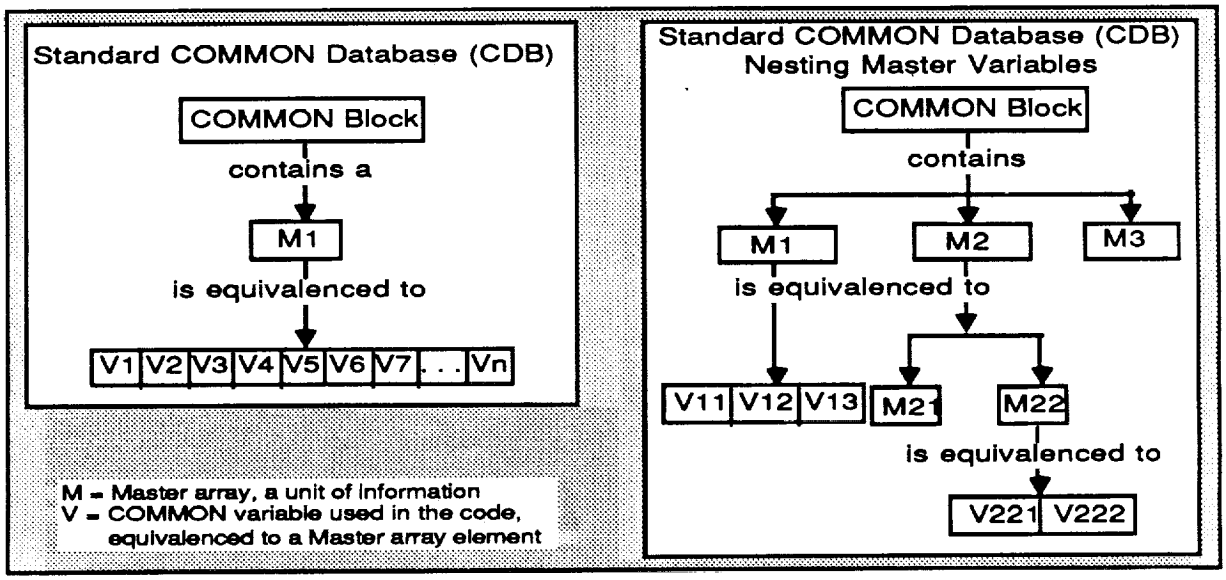

Figure 4. Logically Grouping Data

- Store the subprogram classification information in a module description file. As the code is examined and the purposes of various subprograms are learned, create a description file either external to the code or, preferably, by inserting CD1 comment statements in-line with the code. If CD1 comment statements are being inserted within the code, the subprogram classification information may be stored into a program element table (PETAB) and key word table (KEYTAB) so that the SUBDOC processor can generate meaningful subprogram purpose documentation. (See the appendix for brief descriptions of each of the CD statements, SUBDOC, PETAB table, and

KEYTAB table. For further information, see the *Software Development and Maintenance Aids Catalog* (NASA IN 86-FM-27.)

2.5 Determine the Function of Each Subprogram

After the structures of a subject program are understood, it is necessary to look at the individual program units again and understand *what* each unit does - not how, but what. This will require a lot of reading of the source code. The presence of reliable comment statements will obviously help. Design documents, if available, will help depending on their currency and accuracy. However, the problem with comments and design documents is that there is no way to ensure their correctness. They may be out of date, or they may well have been wrong in the first place. Read them, but remember that a program does what its code says, not what the comments or documents claim it will. The FORREF processor provides cross-reference displays to help to analyze the code of a FORTRAN subprogram. The DDT processor can be used to generate detailed "debug" trace information, if it is necessary to understand particularly tricky code.

3.0 METHOD FOR UPGRADING AN ARBITRARY FORTRAN PROGRAM TO COMGEN COMPATIBILITY

COMGEN-compatibility provides three primary benefits: documentation is provided in structured in-line comments, the data structures are pre-defined, and the source code is compatible with the STB's tool set, which assists a programmer/analyst in maintenance and design recovery activities. This tool set runs most effectively against a program with the COMMON database (CDB). The structure of the COMMON blocks is defined in the CDB and maintained with the aid of various STB processors. The use of a CDB improves the maintainability of the COMMON structure in large FORTRAN programs. The CDB is described in detail in the *Software Development and Maintenance Aids Catalog* (NASA IN 86-FM-27).

Many of the steps in updating an arbitrary FORTRAN program to this standard COMMON structure have been automated by the CCREF, CLEANUP, and SPECPN processors. The upgrade steps are presented in the following paragraphs with particular emphasis on potential problem areas.

The following list of activities summarizes the method for upgrading any arbitrary FORTRAN program to COMGEN-compatibility:

1. Examine the current structure of the COMMON blocks of the program and decide which COMMON blocks to place under control of the standard COMMON concept.

2. Separate the COMMON specification statements from the internal specification statements in each subprogram.

3. Convert the COMMON structure to the EQUIVALENCE format.

4. Achieve variable name uniqueness in the COMMON structure.

5. Create a CDB for the program.

6. Remove unneeded EQUIVALENCE statements.

7. Verify COMGEN-compatibility.

A programmer/analyst may not need to perform every operation discussed in the following steps. If the program COMMON structure is fairly clean, the COMMON upgrade should take fewer steps and may be totally automated.

3.1 Examine COMMON Structure

The programmer/analyst runs the CCREF processor on the subject program to obtain a COMMON block cross reference display of how the COMMON is structured in the program. The CCREF output provides the information necessary to decide which, if any, of the COMMON blocks the programmer/analyst does not wish to include in the CDB. Blank COMMON (used as a program scratch area) should *not* be included in the CDB.

3.2 Separate COMMON Specification Statements from Internal Specification Statements

Upon determining which COMMON blocks are to be included in the CDB, the programmer/analyst runs the CLEANUP processor with the c-option. This operation will reconstruct all specification statements from each subprogram so that all selected COMMON blocks and related specification statements are segregated from specification statements related to local variables and COMMON variables of excluded COMMON blocks. This operation places the specification statements for those COMMONs to be included in the CDB within the standard COMMON delimiting comment statements as follows:

```
C***BEGIN STANDARD COMMON
      .
      .
      .
      Specification statements relative to the CDB
      .
      .
      .
C***END STANDARD COMMON
      .
      .
      .
      Non CDB related specification statements
      .
      .
      .
```

3.3 Convert COMMON Structure to EQUIVALENCE Format

This step to convert COMMON structure to EQUIVALENCE format will be required only if the COMMON in the program being converted is in the in-line type format; i.e., the COMMON statements are of the following form:

```
COMMON /ARRAY/ A, B (3), C
```

To comply with the CDB standards, the only variables that appear in a COMMON statement are "master arrays." Variables equivalenced to locations in a master array are called "common variables." In the actual subprogram code the common variables are referenced. In the example below, ARRAY, is a master array and A, B, and C are common variables. The type of in-line COMMON structure in the example above must be converted to the EQUIVALENCE format shown below by means of the CLEANUP processor with the c-option:

```
      DIMENSION B (3)
      COMMON /ARRAY/ ARRAY (1)
      EQUIVALENCE (A   , ARRAY ( 1) )
     1,       (B   , ARRAY ( 2) )
     1,       (C   , ARRAY ( 5) )
```

CLEANUP performs this upgrade automatically for each COMMON block under standard COMMON control in the program.

3.4 Achieve Variable Name Uniqueness in COMMON Structure

Depending on the structure of the COMMON in the subject program, this step may range in complexity from needing a great amount of work to not being needed at all. The determining factor is how well the subject program adhered to a unique name per location in its COMMON structure. The two standard COMMON rules concerning name uniqueness are as follows:

- The same name cannot appear in two different COMMON locations.
- More than one name can occupy the same COMMON location, but this adds complexity to the COMMON structure and should be limited to use only where necessary.

To begin this process, the programmer/analyst runs the CCREF processor (with the c-option) on the program source code obtained from the last step, which converted COMMON structure to the EQUIVALENCE format. The current step will cause the program to test the code for adherence to the standard COMMON uniqueness rules. The output messages identify the exceptions to the uniqueness standards and the work that needs to be done to the program to complete the process.

Various options exist to resolve the non-unique conditions. The recode processing options available in CLEANUP allow the programmer/analyst to rename a variable in one of the following two ways:

- Rename the variable throughout the subprogram.

- Rename the variable only in the standard COMMON controlled region and generate a second level equivalence of the old name to the new name. For example, a rename of B to A under this option would produce the following structure:

```
C*** BEGIN STANDARD COMMON

      COMMON / NAME / ARRAY(1000)

      EQUIVALENCE (A   , ARRAY( 10)  )

C*** END STANDARD COMMON

      EQUIVALENCE (A   , B)
```

The second standard COMMON rule relating to two variable names in the same COMMON location can be resolved in the following two ways:

- Rename one of the variables to be the same as the other, as discussed above.

- Leave both variables in the CDB. Use of this option should be held to the absolute minimum since it complicates the COMMON structure. If both variables are used in the same subprogram, the format indicated in the example above can be used, or the programmer/analyst can create the following structure.

```
C*** BEGIN STANDARD COMMON

      COMMON / NAME / ARRAY(1000)

      EQUIVALENCE (A   ,  ARRAY(10) )

      EQUIVALENCE (B   ,  ARRAY(10) )

C*** END STANDARD COMMON
```

After performing all the recodes using the CLEANUP processor, run the CCREF processor again to help verify that the results have been as desired.

Care must be exercised when doing recodes on dimensioned variables using the CLEANUP processor. If both the old and the new variable names already appear in the subprogram, they should have the same dimension in order to avoid possible errors because the dimension that will remain is that of the new variable. When doing recodes where the old name and the new name are not equivalenced to each other, the programmer/analyst must be sure that the new name is not already a local variable in the subprogram. If this is the case, then the programmer/analyst must first run a recode to change the name of the local variable to something else. Then the desired recode can be performed, replacing the old name with the new name. Because of the CLEANUP c-option logic, these operations cannot be done in a single execution of the processor.

There are two situations where the programmer/analyst may not wish to do a recode operation, but wishes instead to leave a second level equivalence in the program:

- Equivalences of the following form:

  EQUIVALENCE (B(6), C)

  In this case the CLEANUP processor cannot replace C by B(6); therefore, this EQUIVALENCE will have to remain unchanged.

- Cases where certain names in certain routines have special meaning, but where these are not the names the programmer/analyst wishes to have in the CDB.

3.5 Create a COMMON Database (CDB) for the Program

Once the COMMON structure has been transformed to a state that is compatible with the standard COMMON rules, a CDB can be created for the program by using the CREATE processor. The programmer/analyst will have to make the following two types of modifications to this database to achieve a final form of the CDB:

- The programmer/analyst will have to add the COMMON variable and COMMON array definitions by use of the UPDATE processor.

- If the programmer/analyst has two master arrays in the same COMMON block, the CDB created by CREATE will have these two arrays as two separate COMMON blocks. By using a text editor, the programmer/analyst will have to combine these COMMONs into one CDB COMMON.

3.6 Remove Unneeded EQIVALENCE Statements

The upgrade is now complete except for one possible step that the programmer/analyst may wish to perform. The CCREF processor is run with the c-option. This execution may contain listings of the error message:

VARIABLE "'XXX" IS EQUIVALENCED TO COMMON BUT NOWHERE USED

These unused EQUIVALENCEs are usually the result of the step that converts COMMON structure to EQUIVALENCE format. These unused EQUIVALENCEs can be removed in one of the following two ways:

- If there are only a few unused EQUIVALENCEs, the CLEANUP processor can be used to perform the removal by individually processing each subprogram.

- If there are a large number of unused EQUIVALENCEs, the SPECPN processor removes the unused EQUIVALENCEs from the program source code.

3.7 Verify COMGEN-Compatibility

After a program has been modified to the stage of COMGEN-compatibility, the CAUDIT processor can be run to check the code against the standards.

4.0 METHOD FOR UPGRADING A COMGEN-COMPATIBLE FORTRAN PROGRAM TO NEW "STANDARD" FORTRAN

New FORTRAN standards are proposed in Volume 2 of this document. Upgrading a FORTRAN program to meet these standards will make the program easier to understand, more maintainable, and easier to convert to another language such as FORTRAN 8x, Ada, or C. Many of the steps in updating a COMGEN-compatible FORTRAN program to meet the new FORTRAN standards are supported by tools that exist in the STB's tool set. In the following paragraphs the method to upgrade existing COMGEN-compatible code to meet these standards is described, the existing tools are identified, and brief coding examples are given. The steps are grouped as follows:

1. Rename variables with longer, more meaningful names.
2. Convert to modern control flow structures.
3. Group subprograms into packages.

4.1 Rename Variables with Longer Names

The CLEANUP processor provides the capability to rename variables in FORTRAN source code. This processor will be updated with a front-end that will present the programmer/analyst with all the names found in a given file and ask for new names to be entered. The processor will then perform the renaming to longer, more meaningful names in one pass.

4.2 Convert to Modern Control Flow Structures

The CONVERT processor with the b-option provides for the conversion of old style code to a more structured format. CONVERT cannot remove all GO TO's from general FORTRAN code. In order to remove all GO TO statements, additional non-ANSI standard constructs would be needed, such as CASE and the REPEAT/UNTIL loop, which are not supported by most current FORTRAN compilers. For this reason, CONVERT does not support these constructs. Following are examples of code conversions to more modern control flow structures that the CONVERT processor and most FORTRAN compilers do support.

Example 1

The following code converts a string to upper case characters:

```
NCHAR = LEN (STRING)
DO 100 I = 1,NCHAR
IV = ICHAR (STRING(I:I))
IF (IV .LT. 97) GO TO 100
IF (IV .GT.122) GO TO 100
```

```
      IV= IV - 32
      STRING(I:I) = CHAR(IV)
  100 CONTINUE
```

After processing by the CONVERT processor, the code has the following form:

```
      NCHAR = LEN (STRING)
      DO 100 I = 1,NCHAR
      IV = ICHAR (STRING(I:I))
      IF (IV .GE. 97) THEN
          IF (IV .LE.122) THEN
              IV= IV - 32
              STRING(I:I) = CHAR(IV)
          END IF
      END IF
  100 CONTINUE
```

Example 2

FORTRAN logical IF constructs such as the following:

```
IF (DEBUF .NE. 0) CALL PRINT (A,B,C)
```

are converted by the CONVERT processor to the following structured form:

```
IF (DEBUF .NE. 0) THEN
    CALL PRINT (A,B,C)
END IF
```

Example 3

The CONVERT processor can also create IF-THEN-ELSE structures as shown by the following example:

```
      N = ILCHAR + IFCHAR
      MOVE = JCHAR - N
      KCHAR = NCHAR - ILCHAR
      IF (MOVE .EQ. 0) GO TO 50
      IF (MOVE .LT. 0) GO TO 25
      CALL C4MOVE (CARD(NCHAR),-1,CARD(NCHAR+MOVE),-1,KCHAR)
      GO TO 50
   25 CONTINUE
      CALL C4MOVE (CARD(ILCHAR+1),1,CARD(ILCHAR=1=MOVE),1,KCHAR)
      KCHAR = -MOVE
      CALL C4MOVE (IBLANK,0,CHARD(NCHAR),-1,KCHAR)
   50 CONTINUE
```

This code translates to the following:

```
N = ILCHAR + IFCHAR
MOVE = JCHAR - N
KCHAR = NCHAR - ILCHAR
IF (MOVE .NE. 0) THEN
   IF (MOVE .GE. 0) THEN
     CALL C4MOVE (CARD(NCHAR),-1,CARD(NCHAR+MOVE),-1,KCHAR)
   ELSE
     CALL C4MOVE (CARD(ILCHAR+1),1,CARD(ILCHAR=1=MOVE),1,KCHAR)
     KCHAR = -MOVE
     CALL C4MOVE (IBLANK,0,CHARD(NCHAR),-1,KCHAR)
   END IF
END IF
```

Example 4

The CONVERT processor will also drop the statement numbers from DO loops, thus changing them to the BLOCK DO format as in the following example:

```
      DO 100 I = I,NCHAR
      IV = ICHAR (SSTRING(I:I))
      STRING(I:I) = CHAR (IV)
100   CONTINUE
```

After processing by the CONVERT processor, this code becomes of the following form:

```
      DO I = I,NCHAR
      IV = ICHAR (SSTRING(I:I))
      STRING(I:I) = CHAR (IV)
      END DO
```

Example 5

The CONVERT processor can also convert certain IF structures to DO WHILE loops, as in the following case:

```
100   CONTINUE
      IF (IV .GE.0) THEN
         STRING (IV) = "
         IV =IV - 1
         GO TO 100
      END IF
```

The CONVERT processor will change this to the following form:

```
      DOWHILE (IV .GE.0)
         STRING(IV) = "
         IV = IV - 1
      ENDDO
```

4.3 Group Routines into Packages

A traditional approach to managing the complexity of problems has been to divide large problems into a series of subproblems, each of which is more or less independent of the others. The language construct that historically has helped software engineers to implement this strategy in software has been the subprogram. A new language construct, an extension of the subprogram concept called a *package*, provides a mechanism to split large, complex programs into larger self-contained units made up of logically related subprograms. Using this mechanism, information can be made available to one or more subprograms yet be hidden from the rest of the program. The Ada language directly provides for this capability. The same concept can be applied to *virtual* packages by the programmer/analyst using FORTRAN 77 or C. However, the construction and information hiding of a virtual package created in one of these languages must be done procedurally and this requires programmer discipline.

In Volume 2, which defines the proposed coding FORTRAN standards, a somewhat artificial technique of using comment statements and specification data files to define the virtual packages is described. New PACKAGE and VISIBLE statements are to be added as CD0 comment statements in the prolog of a FORTRAN subprogram in the forms "PACKAGE = package_name" and "VISIBLE = yes or no". Each subprogram in a virtual package will have the same package_name field of the PACKAGE statement. The VISIBLE statement will be equal to "no" for subprograms internal to the virtual package. These subprograms are called only by subprograms with the same value of PACKAGE. VISIBLE will be equal to "yes" if the subprogram is called by subprograms with different values of PACKAGE. A pre-processing tool, i.e., a code auditor, will be developed to help ensure compliance for programmers who use the PACKAGE and VISIBLE mechanism to create virtual packages in FORTRAN programs.

This document is not going to define *how* the subprograms should be organized into virtual packages because the organization of programs can take on a variety of forms. The traditional function structure and the more recent object structure are the two primary approaches. Gurus for each approach address the issue with something approaching a religious zeal. There is an increasing number of proponents for a hybrid system in which each structure is applied where it is the most natural fit (Constantine, 1990). Although standards on this issue are not proposed in this document, it is advantageous to be able to view the structure of a large complex program at a level of abstraction that is higher than that provided at the subprogram level.

In section 2 of this volume, which describes a method for design recovery, ways are identified whereby the existing logical groupings of subprograms in a subject FORTRAN program can be identified using some of the tools in the existing tool set. The programmer/analyst applying the object-oriented approach may start by forming object-oriented packages from groupings of subprograms that are characterized by data cohesion. Someone taking a functional approach may begin by formalizing the logical groupings characterized by control cohesion as packages. The amount of redesign that is done to a subject program is largely determined by this activity, and is a decision to be made by each individual upgrade project.

5.0 METHOD FOR CONVERTING A NEW "STANDARD" FORTRAN PROGRAM TO ANOTHER LANGUAGE

A program that meets the new FORTRAN standards proposed in this document can be translated almost automatically into one of the more modern languages, such as FORTRAN 8x, Ada, or C. The modifications to the structure of data, control flow, subprogram grouping, and meaningfulness of variable names, all of which contribute to successful translation, have already been performed for a program that complies with these new standards. COTS translators are being investigated for inclusion in the proposed environment for redesign and reengineering.

6.0 METHOD FOR CONVERTING A COMGEN-COMPATIBLE FORTRAN PROGRAM TO ANOTHER LANGUAGE

This method is general enough to be used to reengineer a COMGEN-compatible FORTRAN program to either FORTRAN 8x, Ada, or C, incorporating modern software engineering concepts and techniques. The converted program will have more loosely coupled and tightly cohesive modules, and will be more readable and maintainable. The conversion approach offers these benefits at a lower cost in schedule and in application-skilled personnel than redevelopment from scratch. To properly understand the rationale for the the steps of this conversion method, the reader is urged to read Volume 5, which documents the procedures and the lessons learned in the prototype conversion of the Orbital Maneuver Processor (OMP) from FORTRAN to a partially object-oriented Ada. The Orbital Maneuver Processor generally met the CDB standards for COMMON blocks, which simplified the conversion.

The method presented here focuses on grouping the original FORTRAN subprograms into groups of logically related units. In most cases the original subprograms are translated to the target language with no major change to their operation. The only changes in operation are to remove redundancy or to better support the new language and software concepts. The following are some of the advantages of this approach:

- The basic algorithm and logical interface of each subprogram remains largely unchanged, reducing the time needed for the conversion.

- The conversion team members need not be experts in the application domain, although domain expertise helps.

- The logical interfaces of each subprogram will be fairly easy to determine, reducing the usual design concern for completeness.

- Testing is simplified, because the converted unit can be tested side-by-side against the original FORTRAN unit.

The most important software engineering principles to be observed are modularity, localization, abstraction, and information hiding/encapsulation. This conversion method emphasizes these principles. The Ada package concept is of great assistance in meeting these goals. An Ada package is a collection of exported services and/or types with a visible part (the specification) and a hidden implementation (the body). Although packages are referenced in the Ada sense in this document, the concept is meant and it is not intended to imply that only Ada will suffice

even though Ada directly supports the concept. Virtual packages can be implemented in FORTRAN (see section 4.3) or in C, but this requires programmer discipline.

The conversion of a FORTRAN program falls into five basic steps. The first step is design recovery of the FORTRAN program, gaining insight into its purpose and structures. The second step is focussed on an initial partitioning into packages. The primary output of the second step is an initial set of package definitions. The third step is writing detailed specifications for the packages, analogous to writing the interface control documents for the packages. It requires intensive study of the input/output of the FORTRAN subprograms within each package to properly define the inputs and outputs of the reengineered packages. The fourth step is implementation of the package specifications. If the specifications are complete and correct, this step will be the most mechanical of the five. The final step is the unit testing of the converted program.

6.1 Design Recovery

Design recovery has been discussed in detail in section 2 as a separate method because it is fundamental to gaining an understanding of any unfamiliar, large, complex FORTRAN program. This understanding is required to some extent for any maintenance support of a large program, as well as for the first step in converting a program as described in this section and the next.

The product of design recovery may be a document that records the design of the current program. Even if a formal document is not required, comments should be added in-line with the code to help preserve the design knowledge that has been captured.

6.2 Initial Partitioning

Although steps one and two, design recovery and initial partitioning (i.e., redesign), are described sequentially, they are actually performed in parallel. Design recovery is started first, then as understanding increases a notional set of packages to replace the current structure takes form. Once the programmer/analyst has begun to understand the current program, it is time to start to consider the packaging of the converted program. Figure 5 shows the data flow in the steps of design recovery and initial partitioning.

The opportunity to apply object-oriented design is greater in the complete redevelopment of a program or with reverse engineering an existing program to the point of recovering the requirements than with reengineering an existing program to the point of design recovery. The latter case is addressed by this method. The primary definition of object-oriented design used in formulating this method is taken from Grady Booch's *Software Engineering With Ada* (1983). An "object" is defined as something, whether a constant or variable, simple or complex, that maps to some real-world entity. Booch's classification of packages as falling into four types may be helpful: named collections of declarations, groups of related program units, abstract data types, and abstract state machines. (See Booch, 1983 for definitions.) Remember that packages that are object-oriented should provide a "complete" set of operations for the object, so input/output subprograms will also go in the package – at least at a high level.

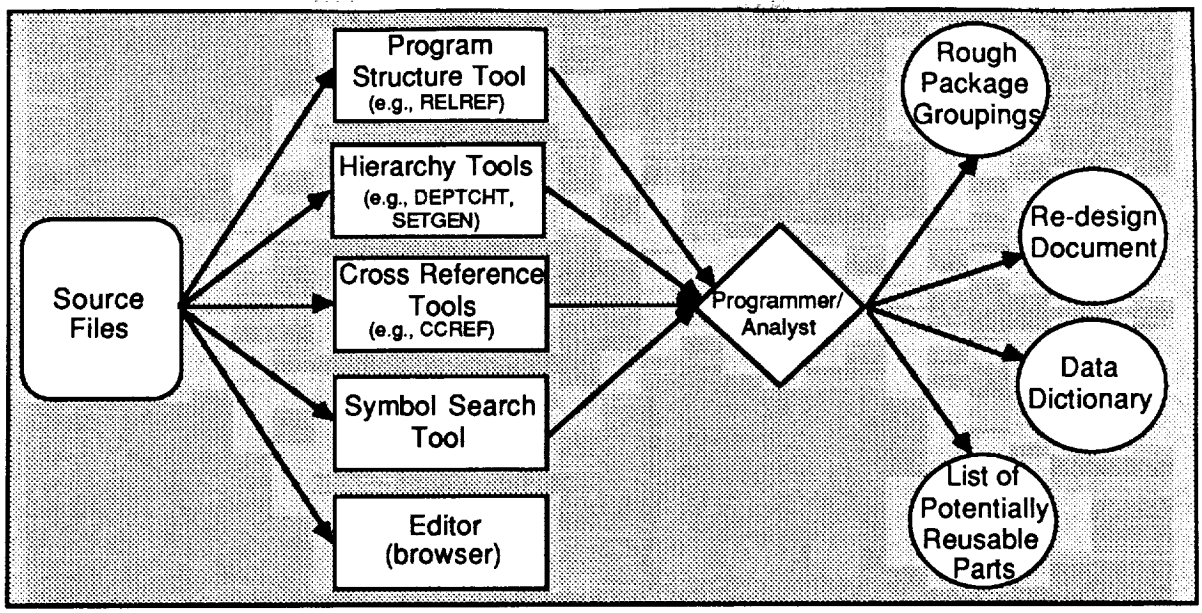

Figure 5. Data Flow in Design Recovery and Redesign Steps

After the object-oriented packages are defined, the remaining packages are largely functionally-oriented. Booch's four classes of packages are still useful. Packages consisting of groups of related subprograms will be the hardest to identify; the best help here is to look at the scope of COMMON blocks and to understand the functions performed by the routines. Some of this will require knowledge of the domain.

The products of the initial redesign may include:

- An initial design document for the converted program which describes each package, using any design method chosen by the conversion team.
- A data dictionary, or description of abstract data types and objects.
- Identification of potential reusable parts, if desired. (This would require more domain knowledge.)

At the end of the initial redesign, there should be a formal review of the first cut at package partitioning. Attendees should be the members of the conversion team and any engineers with knowledge of the original program. The knowledgeable engineers will be able to point out any inconsistencies in the first cut packages and may be able to help identify more potentially reusable packages.

6.3 Package Specifications

Step three, specifying the packages, creates the detailed specifications for the packages that have been identified in the initial redesign. Figure 6 shows the data flow for the package specification step. A package specification is the interface control document (ICD) for the

package, defining exactly what the rest of the program needs to know about the package, but no more. Well-defined package specifications allow a multi-person group to work in parallel on the package implementations.

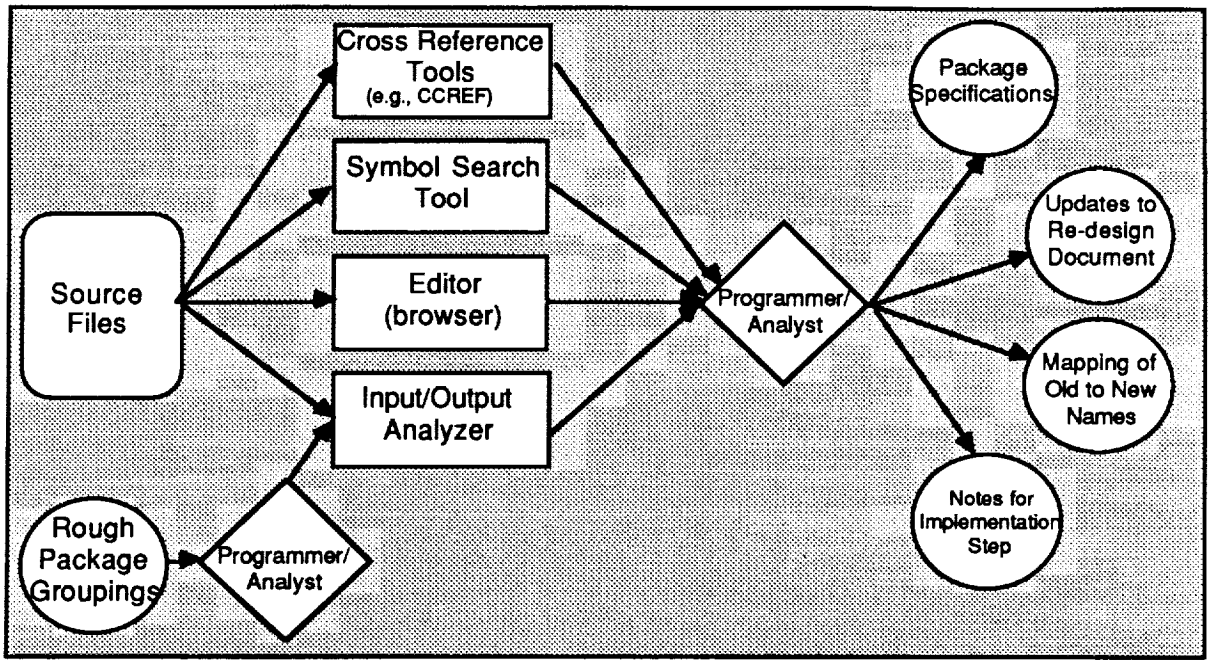

Figure 6. Data Flow for Package Specification Step

The following activities are performed for each package:

- Draw a conceptual line around the package and list all input/output. This includes parameters to all subprograms, COMMON accessed by all subprograms, and calls to other subprograms. The cross-reference tools identified in section 2 can help in this analysis. A package may consist of N separate and independent subprograms, each with its own interface, or N subprograms in a subhierarchy, accessed via a single entry, or a combination of the two. (See figure 7.) Note that data passed to subprogram A must also provide all data for subprograms called by A, unless A generates the data itself or the data will be provided by a low-level request from some other package. This means that, for instance, if subprogram A is the entry point for a package consisting of a group of related subprograms, it is not sufficient to look at subprogram A's inputs and outputs: you must also add the inputs and outputs for all A's subunits. At this point, you are determining *what* information A and its subunits need, rather than the intra-module *how* that is addressed in the implementation step, which is performed later.

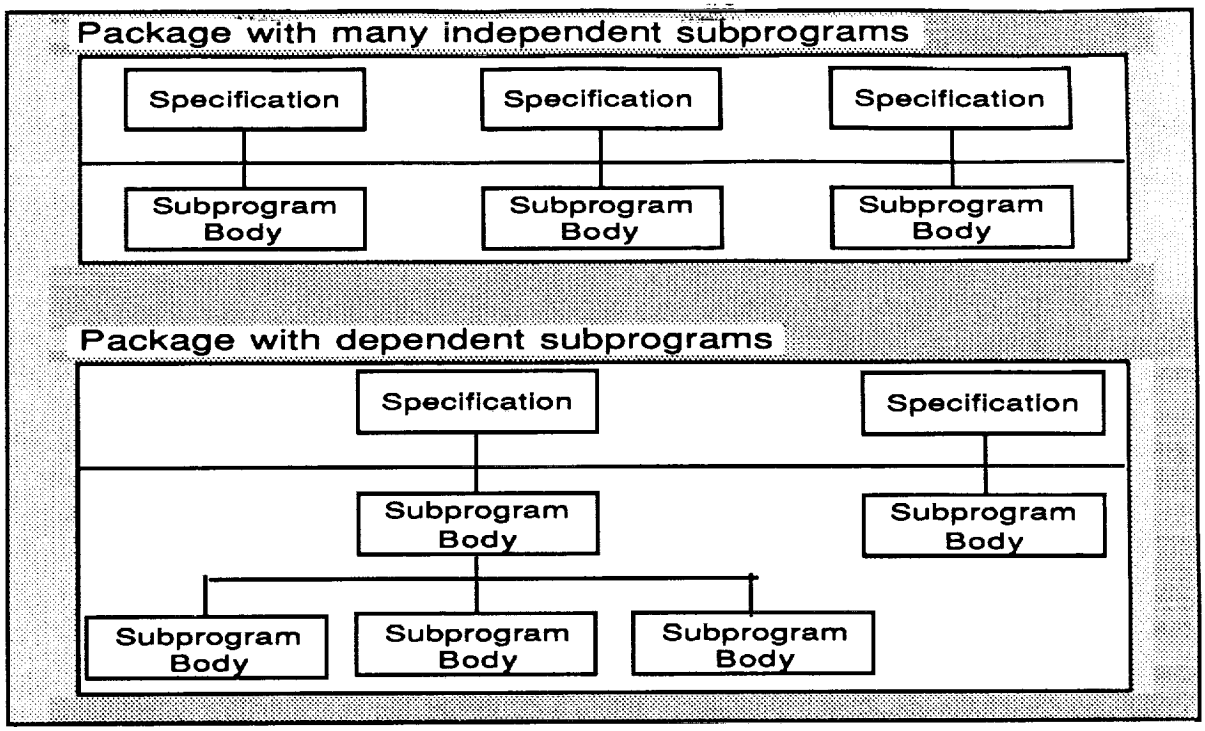

Figure 7. Package Configurations

- Determine the mode (IN, OUT, IN OUT) of all input/output. Look at the terms in a timeline fashion: are they set first or read first? The CREATE processor generates a table containing this information for all the COMMON variables in a subprogram. Is the reference conditional?

- Write a specification describing the provided subprograms and types. The names for the provided subprograms and their parameters should be meaningful and thus usually longer than the original FORTRAN names when possible. In Ada, this will be a package specification with a prolog describing the package functions. In FORTRAN, this might be a set of design notes in comment statements. Current FORTRAN does not syntactically support the equivalent of an Ada specification, but the design notes should be as complete as an actual Ada specification and should be perceived as being binding on the implementer. If changes are made, either later during the package specification or during implementation, the design notes must also be updated. FORTRAN may be syntactically inferior to Ada in this area, but a good procedure, *if followed*, can achieve most of the same benefits.

- Make notes for later use as to whether a subprogram is an exact translation of a FORTRAN subprogram, a partial translation where not all the FORTRAN functionality is necessary, or a new capability. An example is a capability that in the original FORTRAN was provided by accessing a COMMON variable, but in Ada is provided by a subprogram to read or write a value.

Products of the package specification step include the following:

- Complete set of specifications that encompass all the functionality of the original FORTRAN. In Ada, the specifications will be compilable. In C or upgraded FORTRAN, the completeness and correctness of the written specifications must be checked manually.
- Mappings of old names to new names for inter-package variables.
- Updates to the design document describing packages in greater detail and describing interfaces.
- Notes for implementation step.
  - -- Complete mapping between original FORTRAN names and new names (for inter-package communication only) for subprograms, calling parameters, and global entities.
  - -- Notes on which FORTRAN interfaces are to be provided by references in the package body (e.g., base date and tolerances may not be provided as input parameters to some packages because the package body will obtain them via subprogram call to another package).
  - -- Identification of anything that is *not* one-to-one from FORTRAN to the target language.

At the end of the package specification step, there should be a formal review of the design to date. Ideally the attendees will be the conversion team and a few outside senior personnel to consider the correctness of the new design (conformance to standards, completeness, etc.). The conversion team reviews the specifications and the implementation notes.

6.4 Implementation

Step four is the implementation of the package specifications, whether compilable Ada or procedural FORTRAN or C. It is the most mechanical step of the process and could be partially automated by the use of a translator, but there is still a good deal of human intervention required for several reasons. First, some procedures have been slightly changed, and others have been added, so the process is not a straightforward one-to-one translation. Second, the determination of longer and more meaningful names requires human decisions which can only be *assisted* by a tool. Third, there will be some poor code in the original FORTRAN that an automatic tool, or a translator, would be unable to correct. An example is a variable that is described one way, but is actually used to store values with several different meanings. A tool might be able to detect this problem but could not correct it.

The implementation step is the most language dependent. The basic intent is similar, but the details of different target languages will affect the detailed substeps. In this document a conversion from FORTRAN to Ada is discussed, but the procedure would be different only in *syntactic* detail for C or FORTRAN 8x.

For a given Ada specification the implementation is a package body which starts as a copy of the specification. The necessary changes are made to the prolog to convert a package specification to a syntactically correct package body, e.g., PACKAGE -> PACKAGE BODY, types need not be repeated, WITH/USE need not be repeated, parts of the prolog would be redundant if repeated.

At this point, two questions arise. Are the visible program units (those declared in the specification) the only units, or are there more in the package body? (See figure 7.) If there are more units than the visible ones, another input/output analysis like that in the package specification step will be required for intra-package communications. Also, will the functionality change in the conversion? Sometimes the FORTRAN program has redundant code which will be removed, or new functions will added to update the code.

After any intra-module input/output analysis for internal subprograms has been performed and rough internal specifications for them have been written, the next step is to begin translation.

- Copy the FORTRAN source into the package body after the subprogram specification. Move the FORTRAN comment statements on the purpose and functional description up to the package or separate unit prolog for later modification.

- Go through the FORTRAN and use the CONVERT processor to restructure it using constructs such as CASE, IF, THEN, ELSE_IF, ELSE, END_IF, etc. (See section 4.2.) The difficulty of this step depends on the quality of the original code. There will be some cases where early RETURNs are the best way to structure the code; these must be noted in the functional description. In general, any deviances from normal practice should be noted.

- Convert FORTRAN syntax to Ada syntax. A translator will automate most of this activity. The following is a list of examples:

  -- Comments from "C" to "--".

  -- Add ";" to end of each statement.

  -- Change FORTRAN boolean relationals such as ".EQ." to their Ada equivalents.

  -- Change "=" assignments to ":=".

  -- Many initialization statements can be replaced by initial values in the data declarations; because of Ada's elaboration rules this will usually give equivalent results, but be very careful. Remember that program unit local data is not retained but data declared in the package body is retained.

  -- Convert to the new Ada data types: for example a FORTRAN Do-loop to assign one vector to another is now a simple assignment. Another example is records, especially variant records, that replace several logically-related FORTRAN variables.

- Convert FORTRAN symbols to more meaningful names where possible. Note that inter-package symbols, i.e., subprogram names, high-level calling parameters, and many COMMON variables, were already renamed in a previous step. The implementation step is concerned with intra-package and subprogram-local symbol names. The CLEANUP processor will be updated to support this activity.

  -- Calling parameter names are given in the subprogram specification.

  -- COMMON replacements have several cases. Some COMMON variables will have been made into calling parameters. Some (constants, tolerances, etc.) are available either directly or by subprogram call from a different package. Some were used for communication within what is now a single package, and can be replaced by parameters to lower subprograms, or by data declarations in the package body. In most cases, useful long names will have to be generated by investigation unless a CDB or equivalent in-line comments are available.

  -- In a COMGEN-compatible program, local variables usually have a description in the subject subprogram's prolog that can be used to derive a new name. If not, and if investigation does not yield a useful name, the original FORTRAN name may be retained. Incorrect names can be worse than nothing.

  -- Use project-determined guidelines for the length of names. They should be long enough to be "meaningful," but not so long that statements run on forever. This is a judgement call. The data declarations should have comments sufficient to clear up any ambiguity due to reduced name lengths.

- Convert subroutine/function calls. This is complex since the subprogram parameters have probably changed: certainly in name, frequently in number, and often in purpose.

- Make esthetic changes (the function of a "pretty printer" or a translator):

  -- Spacing as appropriate, especially within equations.

  -- Statements should not run-on too many lines (a judgement call).

  -- Use a consistent indentation format.

- Go through the code again and attempt to verify that the code makes sense. Update or correct the comments as understanding permits.

- Update or correct the Purpose/Functional Description in the prolog as your understanding permits.

- Make a final pass to compare the FORTRAN to the Ada and double-check the conversion for typos, etc. Take great care with this stage, the possibilities for slip-ups are endless because of the manual nature of the translation. Provision and use of tools to assist in the implementation will alleviate but not remove this requirement.

The product of the implementation step is compiled packages.

6.5 Unit Testing

In most cases it is necessary only to verify that a given Ada unit performs "identically" to its FORTRAN forebear, where "identically" is application defined. One way is to write drivers to invoke both the Ada and FORTRAN units with identical inputs, and to verify that the output matches. Numeric output must match to the appropriate number of significant digits.

Products of the unit testing step include:

- Test cases and results, including comparison to the original FORTRAN.
- Design document, updated as necessary.

Reviews should be attended by the conversion staff to check readability and maintainability of the code, and to verify the conversion. Conversion staff and outside engineers should review the test cases for correctness and sufficiency.

7.0 METHOD FOR CONVERTING AN ARBITRARY FORTRAN PROGRAM TO ANOTHER LANGUAGE

Converting an arbitrary FORTRAN program to FORTRAN 8x, Ada, or C does not differ from converting a COMGEN-compatible FORTRAN program (described in the previous section) in terms of the required steps. The difference is in the degree of difficulty in performing the required design recovery. The COMMON data structure of the CDB and the structured information in the CD statements of a COMGEN-compatible program makes the understanding of the program easier, and hence enhances the probability of a successful conversion.

The success of the design recovery of an arbitrary FORTRAN program is highly dependent upon both the quality of the documentation and how well the code was modularized in the subject program. A well-structured program with good, current in-line comments can be successfully converted by following the steps in the previous section. However, if a program has all of its global data in a single structure, it will be harder to identify the data structures. If a program has each subprogram in a single file, it will be harder to identify modules. If no documentation is contained in the code in the form of comment statements, it will be very difficult to identify the purpose of each subprogram. Sometimes this information is provided in external documents, but these are seldom kept current as a program evolves.

8.0 CONCLUSION

The methods proposed in this volume have been developed with the intent to provide leverage on the past investments in JSC's FORTRAN programs at a low cost and with a low level of risk. A vast amount of engineering knowledge is embodied in JSC's existing FORTRAN programs, but some have become increasingly difficult and expensive to maintain. The approach presented in this volume is flexible, allowing the management with the responsibility for the maintenance of software systems to selectively choose which programs to upgrade for maintainability and to choose the incremental level of maintainability that is affordable. Maintenance upgrades can be performed incrementally, with improved understandability at each increment.

9.0 APPENDIX

The appendix contains brief definitions of the standardized comment statements, tools in the STB tool set, and data sets created and used by the tools. For further information, the reader is referred to *Automated Software Documentation Techniques* (NASA) or *Computer Program Development and Maintenance Techniques* (NASA IN 80-FM-55).

9.1 Standardized Comment Statements (CDs)

The following is an annotated list of the standardized comment statements (known as CDs) that provide in-line documentation in a COMGEN-compatible subprogram.

CD0 Identification.

Documentation on how to reference the subprogram and documentation identifying those responsible for the subprogram.

CD1 Purpose.

Short documentation of what the subprogram does, not how it does it: topic sentence describes the module; the rest elaborates the module. If possible, the actual requirement paragraph(s) allocated to this module.

CD2 Calling Argument Input.

Documents for each input argument the name, dimension, type, length, and definition, with optional extension for other inputs such as data files.

CD3 Calling Argument Output.

Documents for each output argument the name, dimension, type, length, and definition, with an optional extension for other outputs such as data files.

CD4 COMMON Variable Definitions.

Defines COMMON block variables; usually inserted automatically by the INSDOC processor (see following annotated list of STB tools) from definitions in the COMMON database (CDB). Optional extension for other global data such as data files.

CD5 Internal Variables.

Documents for each internal variable in a subprogram the name, dimension, type, length, definition, with optional extension for other local data such as temporary data files.

CD6 External References.

Documents external data files, external subprograms referenced, subprogram referenced by; normally not used because the information can be obtained from

RELREF processor (see following annotated list of STB tools) or display of the ERTAB, the Elements Referenced Table.

CD7 Functional Description and Method.

Documents the logical flow of subprogram in narrative form.

CD8 Assumptions and Limitations.

Documents major assumptions and limitations inherent in the subprogram.

CD9 Special Comments.

Documents any additional information not included in other structures.

CD10 References.

Provides references to formal external documentation and references to other related subprograms (i.e., See also).

CD11 Keywords.

List keywords that describe the subprogram, to be used in library search for subprogram.

9.2 Tool Set

The following is an annotated list of the existing STB tool set that supports design recovery and reengineering and are referenced in the volumes that make up this document. After this list of tools the contents of the data sets that these tools generate and use are identified.

AUTODOC Automatic Subprogram Documentation Processor.

Generates subprogram documentation for a subprogram based on documentation comment statements (i.e., CD statements) in the source code and information on that subprogram that is in the data sets CDB, SCVTAB, EPETAB, EPTAB, ERBTAB, ERTAB, PETAB, ERTAB, DEPTAB.

CAUDIT Code Auditor.

Checks adherence to the coding standards of the mission planning and analysis domain at JSC and checks spelling in the standard comment blocks using the CDB data set.

CCREF COMMON Block Cross-Reference Program.

Generates the following displays to provide a complete picture of the COMMON structure of the program:

- Names of every COMMON block in each subprogram of the program.

- Names of variables in each location of every COMMON block and subprograms that the variable is in.
- For each master array in a COMMON, names of common variables that are equivalenced to each location of that array.
- Inverse of each of the three previous displays.

CLEANUP Cleanup Processor.

Performs source code cleanup operations to help the programmer/analyst to make the code more understandable, more maintainable, and compliant with the standards of the mission planning and analysis domain at JSC. This includes standardizing the documentation statements (CDs), specification statements, control structure; renaming variables and COMMON locations; and inserting new EQUIVALENCES to COMMON.

COMPARE Symbolic File and Element Comparison Processor.

Displays differences between two source code elements or two of the special data sets used by the STB tool set in form of alters of first file to make it match second.

CONVERT Conversion Processor.

Used for conversion operations to be performed on source code elements.

CREATE Database Creation Processor.

Fully creates some data sets used by the STB tool set, builds data set with fields blank for others. Data sets that can be requested: CDB, CVTAB, SCVTAB, ERTAB, ERBTAB, EPTAB, EPETAB, SCBTAB, CBSTAB,PETAB.

DDT Detailed Debug Trace Program.

Analyzes source code and generates compiled version containing print statements for tracing execution.

DEFINE Documentation Definition Processor.

Supports the insertion of standardized in-line documentation by creating CD statements and CDB variable definitions.

DEPCHT Dependency Chart Generator.

Displays the program structure by drawing a series of hierarchy charts that show the subprogram calling flow for the program or a designated subset of the program using the ERTAB and DEPTAB data sets.

DISPLAY Data Set Display Processor.

Creates labeled displays of the following data sets in a format suitable for documentation: CBSTAB, CDB, CVSTAB, CVTAB, DEPTAB, EPETAB, EPTAB, ERBTAB, ERTAB, KEYTAB, PETAB, SCBTAB, SCVTAB.

DOCGEN Document Generator.

Processes ASCII source files with, page numbering, labeling capability, and some of the word-wrap features of a word processor.

DSDGEN Data Structure Definition Processor.

Previews a data structure that is to be placed into a larger document being built by the DOCGEN processor; also build CD statements if the proper option is selected.

FORREF FORTRAN Cross-Reference Display Processor.

Analyzes a FORTRAN subprogram, generating displays of all the line numbers where each symbol and statement number is referenced.

INSDOC COMMON Variable and Keyword Documentation Insertion Processor.

Inserts the COMMON variable definitions (CD4) and inserts key word statements (CD11) from the CDB into the allocated place in the subprogram prolog.

INVERT Data Set Inverter.

Inverts the following data sets: ERTAB/ERBTAB, SCBTAB/CBSTAB, SCVTAB/CVSTAB.

MANGEN Unix Manual Generator.

Generates a Unix on-line manual entry from CD statements in the main routine of a program.

MAZE Memory Map Analyzer.

Displays the memory map of a program in a readable format, filtering out system routines.

MERGE File Merge Processor.

Merges two data sets into one, using CDB, DEPTAB, and PETAB.

OMNIBUS Omnibus Element Processor.

Converts certain symbol data sets into omnibus elements, which are in turn used by various other processors as part of their input.

QUERY Query Processor.

Allows the user in an interactive mode to search a large data set for individual records using omnibus versions of the following data sets: CBSTAB, CDB, CVSTAB, EPTAB, ERBTAB, ERTAB, SCBTAB.

RELREF Relocatable Element Cross-Reference Program.

Displays lists of the subprograms called by each subprogram in a program file and also a listing of the subprograms that call each subprogram of the file.

SCANPF Control Script Generator.

Generates the control script needed to run repeated executions of any of the other tools against groups of routines.

SETGEN Dependent Element Set Generator.

Generates a list of the names of all subprograms used by a specified routine including those called by routines that it calls, using the data sets EPETAB, ERTAB.

SPECPN Common Variable Specification Statement Generation Processor.

Restructures the COMMON in a program so that it is converted to the desired structure defined by the CDB, using the data sets CDB and SCVTAB.

SUBDOC Subprogram Documentation Processor.

Generates a document containing the purpose of each subprogram in program (taken from the CD1 statements), using the data sets PETAB and KEYTAB.

TABLES Tables Processor.

Subset of DOCGEN, generates tables within a document.

TOCGEN Table of Contents (TOC) Generator.

Generates a table of contents display of the elements in a Unisys program file or the files in a UNIX directory. or compares table of contents for two directories.

UPDATE Database Update Processor.

Updates (inserts, deletes, renames, or modifies) an existing data set, using CDB and PETAB.

VERIFY Database Verification Processor.

Performs verification procedures on some of the data sets.

9.3 Data Sets

The data sets that are used by the STB's tool set can be grouped into three categories: COMMON block data sets, documentation data sets, and program structure data sets. The following list is grouped by category identifying the data set contents that are used by the STB's tools that are referenced in the volumes of this document.

COMMON Block Data Sets:

CDB COMMON Database

CVSTAB COMMON Variable versus Subprogram Table

CVTAB COMMON Variable Table

SCBTAB Subroutine versus COMMON Block Table

SCVTAB Subprogram COMMON Variable Table

Program Structure Data Sets

CBSTAB COMMON Block versus Subprogram Table

DEPTAB Dependency Table

EPETAB Entry Point versus Element Table

EPTAB Program versus Entry Point TAble

ERBTAB Elements Referenced by Table

ERTAB Elements Referenced Table(External Reference Table)

Documentation Data Sets

KEYTAB Keyword Table

PETAB Program Element Table

10.0 GLOSSARY

Acronyms:

CASE	Computer aided software engineering.
CD	CD statement, the in-line documentation standard in the mission planning and analysis domain. For further information, the reader is referred to Computer Program Development and Maintenance Techniques, NASA IN 80-FM-55.
CDB	COMMON database of a program meeting the standard COMMON concept in the mission planning and analysis domain; primary documentation of a FORTRAN program's COMMON structure; used by many of the STB's tools. For further information, the reader is referred to Computer Program Development and Maintenance Techniques, NASA IN 80-FM-55.
COTS	Commercial-off-the-shelf.
FADS	Flight Analysis and Design System.
ICD	Interface control document.
JSC	Johnson Space Center.
MOD	Mission Operations Directorate.
MPAD	Mission Planning and Analysis Division, a former division at JSC.
MSD	Mission Support Directorate, a former directorate at JSC.
STB	Software Technology Branch.

Terms:

arbitrary FORTRAN	FORTRAN program that is not compatible with the COMGEN standards long in place for JSC's mission planning and analysis domain.
COMGEN-compatible	FORTRAN program that is compatible with the COMGEN standards long in place for JSC's mission planning and analysis domain. For further information, the reader is referred to *Computer Program Development and Maintenance Techniques*, NASA IN 80-FM-55.

COMMON database	COMMON database of a program meeting the standard COMMON concept in the mission planning and analysis domain; primary documentation of a FORTRAN program's COMMON structure; commonly referred to as the CDB; used by many of the STB's tools. For further information, the reader is referred to *Computer Program Development and Maintenance Techniques*, NASA IN 80-FM-55.
Common variable	In a COMGEN-compliant program, variables equivalenced to locations in a master array.
design recovery	Reverse engineering, the first step for maintenance or reengineering.
environment	Instantiation of a framework, i.e., an integrated collection of tools, that supports one or more methodologies.
framework	Software system to integrate both the data and the control of new and existing tools; usual components include a user interface, object management system, and a tool set.
FORTRAN 77	ANSI standards for FORTRAN in effect in 1990.
FORTRAN 8x	Future ANSI standards for FORTRAN; expected to be approved and released soon; draft standards have been circulated; unofficially referred to as FORTRAN 90.
forward engineering	Process of developing software from "scratch," through the phases of requirements, design, and coding.
master array	In a COMGEN-compliant program, the only variables that appear in a COMMON statement.
package	"A collection of logically related entities or computational resources" (Booch).
reengineering	"The examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form" (Chikofsky and Cross); combination of reverse engineering and forward engineering.
reverse engineering	"The process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction" (Chikofsky and Cross); the first step for maintenance or reengineering; reverse of forward engineering; process of starting with existing code and going backward through the software development life cycle.

software maintenance	Process of modifying existing operational software while leaving its primary functions intact (Boehm, 1980).
subject program	Program that is being maintained or reengineered.
x-option	Reference to a mechanism of using a letter on the execute statement to indicate the control path to be followed in the execution of a program or processor; used by the Unisys, Unix, and DOS operating systems.
virtual package	Package concept as defined by Booch, but implemented either in Ada, which enforces the concept, or in a language in which the concept must be supported procedurally.

10.0 REFERENCES

Boehm, B. W., *Software Engineering Economics*, Englewood Cliffs, NJ: Prentice-Hall, 1981

Booch, Grady, *Software Engineering with Ada*, Menlo Park, CA: Benjamin/Cummings Publishing Co., Inc., 1983.

Booch, Grady, *Software Components with Ada*, Menlo Park, CA: Benjamin/Cummings Publishing Co., Inc., 1987.

Braley, Dennis, *Computer Program Development and Maintenance Techniques*, NASA IN 80-FM-55, Houston, TX: NASA Johnson Space Center, November 1980.

Braley, Dennis, *Automated Software Documentation Techniques*, NASA, Houston, TX: NASA Johnson Space Center, April 1986.

Braley, Dennis, *Software Development and Maintenance Aids Catalog*, NASA IN 86-FM-27, Houston, TX: NASA Johnson Space Center, October 1986.

Braley, Dennis, "A Software Recovery Methodology," unpublished internal document, Houston, TX: NASA Johnson Space Center, FR51, January 1990.

Braley, Dennis, "FORTRAN Standards for Future Translation and/or Design Recovery," unpublished internal document, Houston, TX: NASA Johnson Space Center, FR51, January 1990.

Braley, Dennis and Allan Plumb, "An Environment for Software Conversion and Code Recovery," unpublished internal document, Houston, TX: NASA Johnson Space Center, FR51, March 1990.

Braley, Dennis, *Maintenance Strategies for Design Recovery and Reengineering: FORTRAN Standards*, Volume 2, Houston, TX: NASA Johnson Space Center, June 1990.

Braley, Dennis and Allan Plumb, *Maintenance Strategies for Design Recovery and Reengineering: Concepts for an Environment*, Volume 4, Houston, TX: NASA Johnson Space Center, June 1990.

Chikofsky, Elliot J. and James H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, January 1990.

Clark, Robert G., Programming in Ada: A First Course, Cambridge: Cambridge University Press, 1985.

Constantine, Larry L., "Objects, Functions, and Program Extensibility," *Computer Language*, January 1990.

Fridge, Ernest M., *Maintenance Strategies for Design Recovery and Reengineering: Executive Summary and Problem Statement*, Volume 1, Houston, TX: NASA Johnson Space Center, June 1990.

George, Vivian and Allan Plumb, *A Method for Conversion of FORTRAN Programs*, Houston, TX: Barrios Technology, Inc., January 1990.